第6章　中断处理与设备驱动程序

本章介绍Linux内核中中断的处理机制及 Linux内核是如何管理系统中的物理设备的。对于中断，尽管大多数的中断处理细节是与体系结构相关的，但内核中同时仍有一些通用的处理中断的机制和接口。操作系统正是通过设备驱动程序为用户隐藏下层硬件设备的细节（如虚文件系统为所有安装的文件系统向上层提供一个统一的视图，而与下层的物理设备无关），而设备驱动程序与中断处理息息相关，因此本章先介绍 Linux内核中中断的处理机制，并基于此对Linux中的设备驱动程序做进一步的介绍。

6.1　中断与中断处理

Linux使用大量不同的硬件去执行不同的任务，例如： Linux用视频设备驱动监视器(monitor)，用IDE设备驱动磁盘。你可以用同步的方式来驱动这些设备，这种方法也就是向设备发送某种操作的请求(如：请求把一块内存写入磁盘中)，然后等待设备完成操作。这种方法虽然能正确工作，但效率非常低，操作系统在等待每个操作完成期间会浪费大量时间。一个更高效的方法是先向设备发送请求，然后做其它更有用的工作。当设备完成操作系统的请求后，它会中断操作系统的运行。使用这种机制，同一时刻可以向系统中的设备发出许多未完成的请求。

现在有支持设备中断当前 CPU运行的硬件。大多数的通用处理器 (如：Alpha)都使用十分相似的中断方式。 CPU的某些引脚与电子线路相连，通过改变电子线路上的电压值 (如：从+5V变为-5V)就可以暂停CPU当前运行的程序，使它转去执行用于处理中断的特殊程序 —— 中断处理例程。这些引脚中有一个还可以与间隔计时器相连，每隔 1/1000秒就能收到一个中断。其它的引脚可以连接到系统中像 SCSI控制器等设备上。

在把中断信号传送到 CPU的一个中断引脚上时，系统经常用中断控制器对设备中断进行分组。这种办法节约了 CPU上的中断引脚，也提高了设计系统时的灵活性。中断控制器有控制中断的屏蔽和状态寄存器。设置屏蔽寄存器某一位可以允许或禁止对应的中断，而状态寄存器用于返回系统中当前活跃的中断。

系统中的某些中断是硬连线实现的，如：实时时钟的间隔定时器是永久连接到中断控制器的第3脚上的；而其它引脚所连接的中断内容却是由插在某个 PCI或ISA插槽上的某种控制卡来决定的。例如中断控制器的引脚 4连接到0号PCI插槽上，而0号PCI插槽可能某天插的是以太网网卡，而另一天是 SCSI控制器。根本问题是每个硬件系统有自己的中断路由机制，所以操作系统必须要能灵活地处理。

大多数现代的通用微处理器按相同的方式来处理中断。当发生硬件中断时， CPU会暂停正在执行的指令转移到内存中的某个地址，在这个地址处或者包含中断处理例程、或者是用于转移到中断处理例程的指令。这部分指令通常在 CPU的中断模式下执行，一般没有其它的中断可以在这个模式下发生。但对某些处理器仍然有特殊的情况，某些 CPU把中断按优先级分类，高级中断可以打断低级的中断。这说明第一级的中断处理代码必须得仔细编写，它经

常要有用于存储CPU处理中断前执行状态的栈(CPU的执行状态包括CPU的所有通用寄存器的值和上下文)。某些CPU有一组只在中断模式下才可见的寄存器，中断处理例程可以用这些寄存器去做大多数的上下文(context)保存工作。

在中断处理完成后， CPU的执行状态被恢复，中断被解除了。 CPU继续执行中断前的工作。把中断处理例程设计的尽可能高效对操作系统来说是非常重要的，而操作系统也不应太频繁或太长时间的阻塞中断的发生。

6.1.1 可编程中断控制器

若不是IBM的PC使用了Intel 82C59A-2 CMOS可编程中断控制器及其后继产品，系统设计者本可以自由选择他们想要的中断体系结构。这个控制器在 PC诞生时就出现了，它可以通过在ISA地址空间众所周知地址上的寄存器来进行编程。现在即使是一个十分先进的 CPU支持逻辑芯片组都会在 ISA存贮空间的相同位置保留有功能等价的寄存器组。而非 Intel的系统(如Alpha AXP系统)可以不受这些体系结构的限制，因而经常使用不同的中断控制器。

图1-6-1画出了两个链接在一起的 8位中断控制器。每个控制器都有一个名为 PIC1的屏蔽寄存器和一个名为PIC2的中断状态寄存器。两个中断屏蔽寄存器放在0X21和0XA1的地址处，两个状态寄存器分别在 0X20和0XA0处。向中断屏蔽寄存器的某一位写入 1允许中断，而写0就禁止中断。因此向中断屏蔽寄存器的第 3位写入1能够允许中断3，而写入0即禁止中断3。很令人苦恼的是中断屏蔽存器是只写的，你无法读取刚刚写入的值。这使得 Linux不得不保留一个中断屏蔽寄存器的本地副本，每次在中断允许和中断禁止例程中先改变副本的值，然后再把副本完全写入中断屏蔽寄存器中。

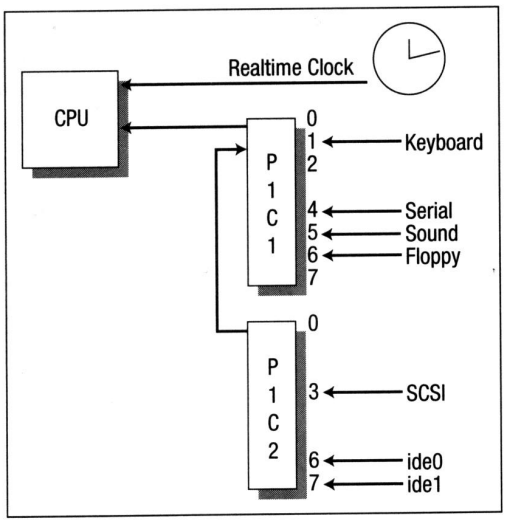


图1-6-1 中断路由逻辑图

中断产生时，中断处理例程读取两个中断状态寄存器的值。它把 0X20处的中断状态寄存器的值放在 16位中断寄存器的低 8位，而把0XA0处的中断状态寄存器的值放在高 8位。因此在0XA0处中断状态寄存器的第一位的中断会被当作系统的第 9位中断。 PIC1的第2位是不可用的，因为它用于连接来自 PIC2的中断的，而PIC2上的任何一个中断都会使 PIC1的第2位置1。

6.1.2 初始化中断处理数据结构

内核的中断处理数据结构是由设备驱动程序在请求系统中断控制时建立起来的。为了建立这些结构，设备驱动程序使用 Linux内核中的一组用于请求中断、允许中断、禁止中断的服务。每个设备驱动程序调用这些例程注册它们的中断处理例程的地址。

有些中断由于遵守 PC体系结构的习惯而被固定下来，因此这种类型的驱动程序在初始化时只是简单地请求它的中断，这正是软盘设备驱动程序所做的 —— 它总是请求IRQ 6。在某些

情况下设备驱动程序无法知道设备使用哪个中断；这对 PCI设备驱动程序是不成为问题的。因为驱动程序总是知道它们的中断号是多少。而让 ISA设备驱动程序找到它们的中断号就不是一个简单的问题。Linux通过允许设备驱动程序检测它们自己中断的方式解决了这个问题。

设备驱动程序先对设备进行某些操作，使设备产生中断。然后它把系统中所有未被分配的中断置成允许状态。这个操作表示该设备的待处理的中断会通过可编程中断控制器传送过来。Linux读中断状态寄存器并把它的值返回给设备驱动程序，一个非 0的值表示在检测期间有一个或多个中断发生了。驱动程序接下来关闭检测并把所有未分配的中断置成禁止状态。如果ISA设备驱动程序成功地找到了它的 IRQ号，那么就可以像正常的驱动程序一样请求中断控制。

基于PCI的系统比基于ISA的系统有更多的动态性。 ISA设备使用的中断引脚是由硬件设备上的跳线来设定的，并且在设备驱动程序中是固定的，而 PCI设备是在系统启动时 PCI初始化过程中由PCI BIOS或PCI子系统来分配中断号的。每个 PCI设备可以使用A、B、C、D四个中断引脚中的一个。在设备生产时，设备的中断引脚就是固定的了，而且大多数设备缺省使用引脚A上的中断。每个PCI插槽上的PCI中断线A、B、C、D都被路由到中断控制器上。所以PCI插槽4上的引脚A就可能路由到中断控制器的引脚 6，而该插槽的引脚 B被路由到中断控制器的引脚7，其它的引脚就以此类推。

PCI中断如何路由完全是由系统决定的。系统中有一些建立程序用于掌握 PCI中断路由的拓扑结构。在基于 Intel的PC上这些建立程序是系统启动时运行的 BIOS中的代码，而对像 Alpha AXP那样没有BIOS的系统，由Linux内核做这部分建立工作。 PCI建立程序把每个设备的中断控制器的引脚号写入到它的 PCI配置头中。它使用 PCI中断路由拓扑结构信息， PCI设备的插槽号以及PCI设备使用的中断引脚号来共同决定该设备的中断号。一个 PCI设备使用的中断引脚是固定的，并记录在该设备的 PCI配置头的一个域中。 PCI建立程序把中断号记录在专门为中断保留的中断线域中，当设备驱动程序运行时，它从配置头中读取中断号信息，并使用它向Linux内核请求中断控制。

在使用PCI-PCI桥时，系统中会有很多 PCI中断资源，因此中断源的数量可能会超过系统的可编程中断控制器的引脚数。在这种情况下， PCI设备要共享中断，即中断控制器的一个引脚要接受一个以上 PCI设备的中断请求。 Linux通过允许中断源的第一个请求者声明该中断是否是可以共享的来支持中断共享机制。共享中断使得 irq_action vector向量中的一项指向若干个irqaction数据结构。当一个共享中断发生时， Linux会调用该中断源的所有中断处理例程。因此，任何支持共享中断的设备驱动程序必须为其中断处理例程被调用而没有中断等着处理这种情况做好准备。

6.1.3 中断处理

Linux中断处理子系统的一个基本任务是把中断路由到正确的中断处理程序去。因此这部分代码必须懂得系统的中断拓扑结构。例如对中断控制器引脚 6上发生的软盘控制器中断，中断处理子系统必须识别出该中断来自于软盘控制器并把它传送给软盘设备驱动程序的中断处理例程。 Linux使用一组指针来指向包含处理系统中断的例程地址的数据结构。这些例程属于系统设备的设备驱动程序，并由设备驱动程序在初始化时请求这些例程使用的中断控制。在图1-6-2中，irq_action是一组指向irqaction数据结构的指针，每个 irqaction数据结构包含关于

该中断处理例程的信息，其中有中断处理例程的地址。由于中断的数量和处理方式随体系结构和系统的不同而不同，所以Linux中断处理程序是与体系结构相关的，这就意味着 irq_action vertor向量的大小取决于系统中中断源的数目。

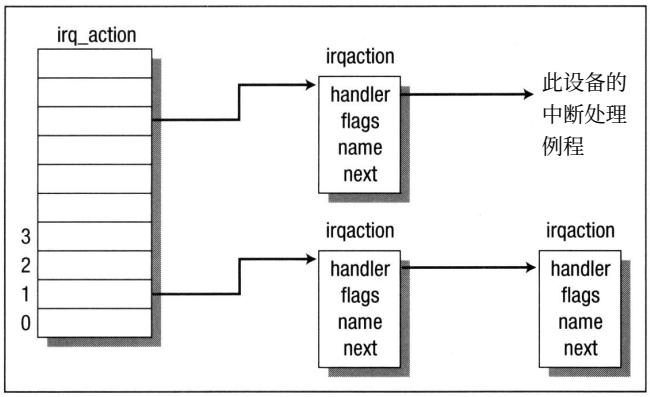


图1-6-2 Linux中断处理数据结构

当中断发生时，Linux通过读取系统的可编程中断控制器中中断状态寄存器的值来确定中断源。然后Linux把中断源转换成相对于irq_action vector向量的偏移。如对在中断控制器引脚6的来自软盘控制器的中断，Linux会把它转换为中断处理向量的第七个指针。如果系统对发生的中断没有对应的中断处理例程，那么 Linux内核会记录一个错误信息。否则的话它会调用该中断源对应的所有irqaction数据结构中的所有中断处理例程。

在设备驱动程序的中断处理例程被 Linux内核调用时，它必须立即确定中断原因并做出响应。为了查找中断原因，设备驱动程序会读取发出中断的设备的状态寄存器，而设备可能报告一条错误信息或是请求操作完成的信息。如软盘控制器可能会报告它已把软驱的磁头定位在软盘的正确扇区上了。一旦中断原因被查明了，设备驱动程序可能需要做进一步的处理工作。如果设备驱动程序要做进一步的工作，Linux内核提供一种允许驱动程序延缓处理工作的机制，它可以避免 CPU在中断模式下花费太多的时间。要想了解进一步的信息，请看 6.2节“设备驱动程序”。

6.2 设备驱动程序

CPU并不是系统中唯一的智能设备，每个物理设备都有它自己的控制器，键盘、鼠标、串口的控制器是SuperIO芯片，IDE磁盘的控制器是IDE控制器，SCSI磁盘的控制器是SCSI控制器……每个硬件控制器都有自己的控制和状态寄存器组 (CSR)并随设备的不同而不同。如Adaptec 2940 SCSI控制器的CSR就完全不同于NCR 810 SCSI控制器的CSR。CSR主要用于启停设备、初始化设备以及诊断设备的故障，Linux并不是把系统中的硬件控制器的管理程序放在应用程序中，而是把这些程序全放在内核里。设备驱动程序指用于处理、管理硬件控制器的软件。Linux内核中的设备驱动程序是一组长驻内存具有特权的共享库，也是一组低级的硬件处理例程。系统正是用Linux的设备驱动程序处理它所管理设备的特殊性问题。

UN*X的一个基本特征就是它抽象了设备的处理。所有的硬件设备都与常规的文件十分相似，它们可以通过与操纵文件完全一样的标准系统调用来打开、关闭、读和写。系统中的每个设备由一个特殊设备文件来表示，如系统中的第一个 IDE硬盘由/dev/hda文件表示。对于块

设备和字符设备，这些特殊设备文件可以由 mknod命令创建，并由主次设备号来描述对应的设备。网络设备也可以由特殊设备文件来表示，但它是在 Linux查找初始化网络控制器时建立的。所有由同一个设备驱动程序驱动的设备有相同的主设备号，次设备号用于把这些不同的设备及它们的控制器区别开。如主 IDE硬盘的每个分区有不同的次设备号。所以 /dev/hda2是主IDE硬盘的第二个分区，它的主设备号为 3，次设备号为2。Linux通过用主设备号和一组系统表格(如：字符设备表— chrdevs)，在系统调用中把特殊设备文件 (假定在块设备的装配文件系统中)映射到设备的设备驱动程序上。 Linux支持三种硬件设备类型：字符设备、块设备、网络设备。字符设备是支持无缓存读写的设备，如系统的串口 /dev/cuao和/dev/cual。块设备只能按多个块的大小进行读写，典型的块大小是 512字节或1024字节。块设备是通过缓冲区缓存来访问的，并支持随机地访问 —— 即无论该块在设备的何处，都能够直接读写。块设备能通过特殊设备文件来访问，但大多数情况下是通过文件系统来访问的。只有块设备才支持安装的文件系统。网络设备通过 BSD套接字接口来访问的，在第 8章网络中对网络子系统有详细的介绍。

Linux内核中有许多不同的设备驱动程序，但它们有一些共同的属性：

内核程序　设备驱动程序是内核的一部分，像其它内核中的程序一样，如果出错可能会严重地损害系统。一个编写得很差的驱动程序甚至会使系统崩溃，也可能损坏文件系统而造成数据丢失。

内核接口　设备驱动程序为Linux内核或内核的子系统提供了一套标准的接口。例如：终端驱动程序为Linux内核提供了文件I/O接口，而SCSI设备驱动程序为SCSI子系统提供了SCSI设备接口，又为内核提供了文件I/O和缓冲区缓存接口。

内核机制和服务　设备驱动程序可以使用像存储分配、中断转接、待操作队列这些标准内核服务。

可加载性　大多数的Linux设备驱动程序可以像内核的模块一样在需要时载入内存，在不使用时卸载，这种方式使得内核在处理系统资源方面适应性强、效率很高。

可配置性　Linux设备驱动程序可以安装到内核中，在内核被编译时，这些内核中的设备驱动程序是可配置的。

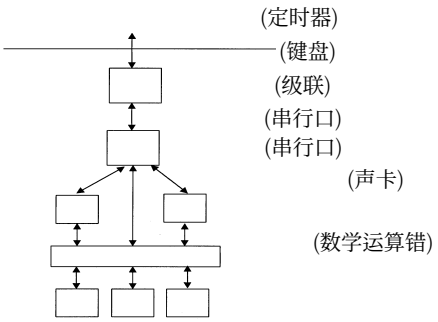
动态性　系统启动时每个设备驱动程序初始化，查找它所控制的硬件设备。如果某个设备驱动程序对应的设备不存在的话，那么该设备驱动程序就变成冗余的驱动程序，除了占用一点系统存储空间以外，不会造成任何损害。

6.2.1 测试与中断

每当设备接收一个类似于“ 将读磁头移到软盘的第 42个扇区 ”的命令时，设备驱动程序可以有两种方式来确定命令的完成情况：不断地测试设备或等待设备的中断。

测试设备表示驱动程序不断地读设备的状态寄存器，等待设备状态寄存器的状态变为操作完成状态。作为内核一部分的设备驱动程序，如果不断测试设备的话，会使得内核在设备完成请求之前无法运行其它的程序。使用测试方式工作的设备驱动程序还可以用系统定时器，让内核在一定时间间隔之后调用设备驱动程序中的一个例程，该定时器例程被调用时会检查设备的命令状态。这种方式是 Linux软盘驱动程序的工作机制。采用定时器的测试机制近乎于是一种最佳的工作机制。但使用中断仍是一种比它更有效的一种方法。

中断驱动的设备驱动程序是指该驱动程序控制的硬件设备在需要服务时，会向驱动程序发硬件中断。例如对一个以太网网卡设备，它在收到网络上的以太网报文时会向以太网驱动程序发中断。Linux内核要具有把来自硬件设备的中断转交给正确的设备驱动程序的能力。这是由设备驱动程序向内核登记它的中断使用情况来实现的。可以通过查看 /proc/interrupts来确定设备驱动程序使用中断的情况，以及当前系统中有多少种中断类型：



对中断资源的申请要在驱动程序初始化时完成。但系统中的一部分中断是固定的，它是IBM PC机体系结构中遗留下来的。如：软盘控制器永远使用中断 6。其它的中断与来自于PCI设备的中断一样是在启动时动态分配的。在这种情况下，设备驱动程序在请求中断拥有权之前必须先找到它所控制的设备的中断号。对 PCI设备的中断， Linux支持标准的PCI BIOS回调函数来检测包括中断号在内的系统中的设备信息。

中断传送到CPU的方式是与体系结构相关的，但在大多数体系结构中中断是通过先阻止系统产生其它中断，这种特别的方式来传送中断的。设备驱动程序应该在中断处理例程中做尽可能少的工作，以使得 Linux内核尽可能快地解除中断，继续执行中断前的任务。那些收到中断后需要做大量处理的驱动程序可以使用内核的 bottom_half处理例程或任务队列机制，将处理例程加入到队列中，使得它在不久后能够被调用。

6.2.2 直接存储器访问(DMA)

在数据量少的时候，使用中断驱动的设备驱动程序向硬件设备传送数据或从硬件设备读取数据工作得比较好。如一个 9600bps的调制解调器可以大约每 1/1000秒传送一个字符。如果中断延迟 —— 从硬件产生中断到设备驱动程序的中断处理例程被调用之间的时间开销，比较低的话(如2/1000秒)，那么数据传输占用的系统的开销就比较低。 9600bps 调制解调器的数据传输大约只花费CPU 0.002%的处理时间。但对像硬盘控制器或以太网设备这些数据传输率高的设备，一个SCSI设备就可以每秒传输40M字节的信息。

DMA(Direct Memory Access，直接存储器访问)就是用于解决这个问题的。 DMA控制器可以在不打扰 CPU的情况下,允许设备将数据传输到存储器中或从存储器中读取数据。一个ISA的DMA控制器有8个DMA通道，其中有7个对设备驱动程序是可用的。每个 DMA通道都有一个16位的地址寄存器和一个 16位的记数寄存器。在初始化数据传输时，设备驱动程序先设定DMA通道的地址寄存器、记数寄存器以及数据传输的方向 —— 读还是写；然后它通知设备可以在就绪后启动 DMA传输了。传输完成时，设备会中断 CPU。在整个传输过程中， CPU完全不受影响可以做任何其它工作。

设备驱动程序在使用 DMA时必须小心：第一点是所有的 DMA控制器根本不知道虚存，它

只能访问系统中的物理内存。而且 DMA传入或传出的数据只能是连续的物理内存块，这就表明你无法对进程的虚地址空间使用 DMA操作；但在DMA操作期间，可以锁定进程的物理页面，防止它被交换到交换设备上去。第二点是 DMA控制器无法访问整个物理内存， DMA通道的地址寄存器代表DMA地址的前16位，而后面8位取自于页面登记表。这说明 DMA请求被限制在下面16M的存储器中。

由于DMA通道只有7个，而且它们不能被设备驱动程序共享，所以这些通道是稀缺的系统资源。就像中断一样，设备驱动程序必须能确定出它们正使用的 DMA通道。某些设备使用固定的DMA通道，如软盘设备永远使用 DMA通道2。有时设备的DMA通道可以由跳线设定，一些以太网设备采用的就是这种技术。更加灵活的设备可以通过设定它们的 CSR来确定使用哪个DMA通道。这时设备驱动程序可以选择一个空闲的 DMA通道来使用。

Linux使用dma_chan数据结构(每个DMA通道对应一个)的向量来跟踪DMA通道的使用情况。dma_chan数据结构包括两个域：一个是指向描述 DMA通道使用者的字符串的指针 ,另一个描述DMA通道是否被占用的标志域。当你使用 cat/proc/dma命令时，屏幕上打印出来的正是dma_chan数据结构的向量。

6.2.3 存储器

设备驱动程序在使用存储器时必须要小心。作为 Linux内核的一部分，它们无法使用虚存。每次设备驱动程序运行时，可能由于收到了中断或者是由于 bottom_half例程或任务队列被调度运行了，因而当前的进程可能会改变。尽管设备驱动程序独立地执行，但它也不能依赖于正在运行的某些特定进程。像内核中的其它进程一样，设备驱动程序用数据结构来跟踪它所操纵的设备。这些数据结构作为设备驱动程序代码的一部分是静态分配的。但由于它使得内核比需要的要大，所以比较浪费。大多数设备驱动程序使用内核中未分页的存贮器来记录数据。

Linux提供了内核存储空间的分配和释放例程，以供设备驱动程序使用。尽管设备驱动程序需要的空间可能不是 2的幂次方，但内核的存储空间必须按 2的幂次方进行分配，如：它可能是128或512字节。设备驱动程序需要的存储空间字节数被扩大到最接近的一个 2的幂次方块，这种方法使内核在释放存贮空间时很容易把小的自由块组合成大的块。

在分配内核存储空间时， Linux需要做大量的额外工作。如果系统中自由空间的数量比较少，系统需要释放一些物理页，并把它写到交换设备上。一般来说， Linux会挂起请求者，把进程放在等待队列上，等待系统出现足够的物理内存。并不是所有的设备驱动程序都想按此处理，所以内核存储空间分配例程在无法立即分配内存时，可以返回失败信息。如果设备驱动程序要对分配的内存空间进行 DMA操作，就可以指出该存贮空间是可 DMA操作的。这正是Linux内核而不是设备驱动程序来获知系统中哪些内存是可 DMA操作的一种方式。

6.2.4 设备驱动程序与内核的接口

Linux内核可以按照一种标准的方式和驱动程序进行交互。每类设备驱动程序 —— 字符设备、块设备、网络设备，都为内核在使用它们的服务时提供相同的使用接口。这些相同的接口使得内核可以对完全不同的设备和它们的驱动程序按照完全相同的方式进行处理，如对SCSI硬盘和IDE硬盘这两个不同的设备来说， Linux内核对它们使用完全相同的接口。

Linux具有很高的动态性，每次 Linux内核启动时，会遇到不同的物理设备，需要不同的设备驱动程序。Linux允许在编译内核时通过配置脚本把设备驱动程序加入到内核中，而这些驱动程序在启动初始化时，允许找不到要控制的硬件。其它的驱动程序可以在需要时作为内核的模块被载入。为了实现设备驱动程序的动态性，设备驱动程序在初始化时要向内核注册。 Linux维护一个设备驱动程序的表，并把它作为与驱动程序接口的一部分。这些表包括支持该类设备接口的例程和其它信息。

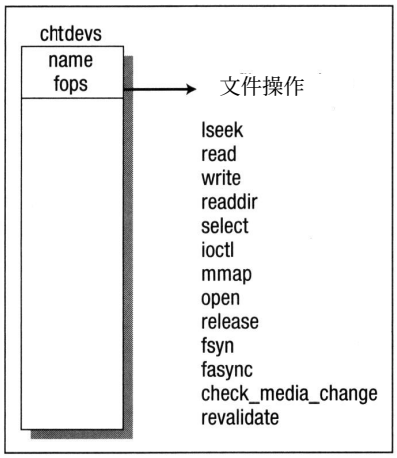


图1-6-3 字符设备

1. 字符设备

字符设备是一种可以像文件一样进行访问的简单的Linux设备(见图1-6-3)。应用程序可以像对待文件一样对待这类设备，用标准的系统调用打开、读、写、关闭它们；即使这类设备可能是 PPP守护程序使用的、用于把Linux系统连接到网络上的modem。但对该设备的这些操作仍然是完全可以的，字符设备的驱动程序通过在 device_struct数据结构的chrdevs向量中增加一项的方法来向Linux内核注册自己。在注册过程中，驱动程序初始化字符设备，该设备的主设备号 (对于tty设备为4)是chrdevs向量的索引值，因此每个设备的主版本号是固定的。在 chrdevs向量的每一项中，device_struct数据结构包括两个部分：一个是指向注册设备驱动程序名字的指针；另一个是指文件操作块的指针。文件操作块中有字符设备驱动程序的处理例程的地址，这些处理例程是用于处理像打开、读、写、关闭这类专门的文件操作。 /proc/devices中关于字符设备的内容是从chrdevs向量中获取的。

在打开一个代表字符设备的特殊设备文件时，内核要做一些建立工作，以使得系统能调用正确的字符设备驱动程序的文件操作例程。像普通的文件和目录一样，每个特殊设备文件由一个VFS inode节点来表示。每个特殊字符设备文件的 VFS inode节点，实际上对所有的特殊设备文件是通用的。它包含设备的主标识和次标识。该 VFS inode节点由下层文件系统创建，在查找特殊设备文件名时，系统会从真正的文件系统中读取关于该设备的信息。

每个VFS inode节点都是与一组文件操作相关联，但这些操作取决于该 inode节点代表的文件系统中的对象。一旦一个代表特殊字符设备文件的 VFS inode节点被建立起来，则它的文件操作被置成字符设备的缺省操作 —— 只有一个打开文件的操作。应用程序打开特殊字符设备文件时，通用的打开文件操作把设备的主标识作为 chrdevs向量的索引，取出该设备的文件操作块。同时它还会为该特殊设备文件建立文件数据结构，把文件数据结构的文件操作指针指向设备驱动程序的文件操作例程。至此所有的应用程序的文件操作都被映射到对字符设备文件操作例程的调用上去了。

2. 块设备

块设备也支持文件操作，为打开的特殊块设备文件提供的对应的文件操作机制与字符设备十分相似。 Linux在blkdevs向量中维护着注册的块设备集，而 blkdevs向量像chrdevs向量一样，由设备的主设备号进行索引。它的每个表项仍然是 device_struct结构，但这些结构是属于块设备的。 SCSI设备是块设备中的一类，而 IDE设备是另一类。正是这些类向 Linux内

核注册自己，并对内核提供文件操作。每一类块设备的驱动程序都为该类提供专门的接口。例如SCSI设备为SCSI子系统提供接口，而 SCSI子系统用该设备提供的文件操作为内核服务。

和正常的文件操作接口一样，每个块设备驱动程序要为缓冲区缓存机制提供接口。每个块设备要填写blk_dev_struct数据结构的blk_dev向量中的对应表项，对该向量的索引仍然是该设备的主设备号。blk_dev_struct数据结构包括请求例程的地址和指向 request数据结构表的指针。每个request数据结构对应于一个缓冲区缓存对该设备读 /写数据块的请求。

每当缓冲区缓存机制要从注册的设备读一块数据或写入一块数据时，它都会在blk_dev_struct中加入一个请求数据结构。图 1-6-4表明每个读写一块数据的请求都有一个指向一个或多个buffer_head数据结构的指针。buffer_head数据结构是由缓冲区缓存锁定的，系统中有等待该块操作完成的进程。每个请求结构是从名为 all_requests的静态表中分配出来的。如果有请求被加到一个空请求表中，该驱动程序的请求函数就会被调用，开始处理请求队列；否则的话，驱动程序会处理请求表中的每个请求。

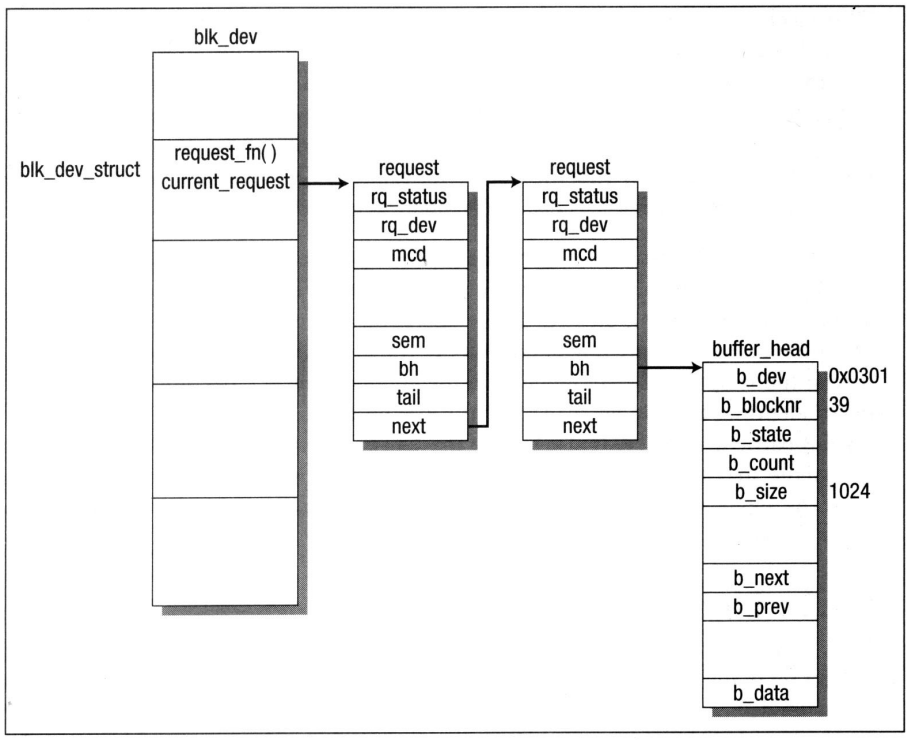


图1-6-4 缓冲区缓冲块设备的请求

一旦设备驱动程序完成了请求，它从该 request结构中删除所有的 buffer_head数据结构，把它们标志为更新状态并解锁。对 buffer_head的解锁会唤醒“睡眠”在等待块操作完成队列上的所有进程。上面过程的一个例子是文件名的解析过程， EXT2文件系统要从包含该文件系统的块设备上读取包含下一级 EXT2目录项的数据块，进程会“睡眠”在包含该目录项的buffer_head结构上，等待设备驱动程序将其唤醒。完成请求后， request被标志为空闲状态，以便于其它块请求可以使用它。

6.2.5 硬盘

硬盘通过把数据记录在盘片上，提供了一种更持久的信息保存方法。写数据时，一个小磁头磁化盘面上的微小质点，接着一个读磁头读出刚写入的数据，以确定某个微小质点是否被正确的磁化。

磁盘驱动器包括一个或多个盘片，每个盘片由光滑的玻璃或复合陶瓷组成，外表覆盖了一层氧化钢。盘片的中央都连接在一个轴上并按恒定的速度转动。转动速度根据硬盘类型的不同从每分钟3000转到每分钟10000转不等。而软盘驱动器的转速只有 360RPM(转/分钟)。硬盘的读写头用于读写数据，每个盘片有一对读写头，一面有一个头。读写头实际上并不接触盘面，它们漂浮在非常薄的 (大约14万分之一英寸)的气垫上。所有的读写头是连接在一起的，在马达的带动下，一起在盘片上移动。

每个盘片的表面被化分成狭窄的同心圆——磁道。 0磁道是最外面的磁道而编号最高的磁道最靠近中轴。柱面指具有相同磁道号的所有磁盘的集合。因此磁盘中所有盘片的所有盘面的第5磁道被称为第5柱面。由于柱面数与磁道数相同，因此你经常可以看到用柱面表示的磁盘结构。每个磁道被划分为扇区，扇区是硬盘读写的最小数据单元，它正好是磁盘块的大小，一般扇区的大小是512字节，扇区的大小通常是在硬盘生产过程中设定的。

硬盘通常由盘面号、头号和扇区号来表示。例如在启动时 Linux把一个IDE硬盘描述为：

```
hdb: Conner Peripherals 540MB - CFS540A, 516MB w/64kB Cache, CHS=1050/16/63
```

这表示它有1050个柱面，16个头，每道有63个扇区。如果扇区大小为512字节，整个盘的存贮容量为529200字节，与磁盘标明的516M容量不符，原因是有些扇区用于存放磁盘分区信息。部分硬盘可以自动地查找坏扇区，并重新建立硬盘索引以跳过这些部分。

硬盘可以进一步地被分成硬盘分区。一个分区指一大组用于某种特殊目的的扇区。对硬盘进行分区使得硬盘能够被多个操作系统使用。大多数的 Linux系统的一个硬盘，有三个分区：一个是DOS文件系统分区；一个是EXT2文件系统分区；最后一个是交换分区。硬盘的分区由分区表来描述，分区表中的每一项用头、扇区、柱面的形式标出分区的开始、终止位置。对DOS格式的硬盘来说，它可以有4个主磁盘分区，由fdisk命令来划分。但在分区表中并不是所有的四个项都是可用的。 fdisk只支持三种类型的分区 —— 主分区、扩展分区、逻辑分区。扩展分区不是真正的分区，它可以包含任意多个逻辑分区。扩展分区加逻辑分区是一种避免四个主分区限制的方式。接下来是 fdisk对一个包含两个主分区的硬盘的输出信息。

```
Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders
Units = cylinders of 2048 * 512 bytes

   Device Boot    Begin     Start       End    Blocks   Id  System
/dev/sda1             1         1       478    489456   83  Linux native
/dev/sda2           479       479       510     32768   82  Linux swap

Expert command (m for help): p

Disk /dev/sda: 64 heads, 32 sectors, 510 cylinders

Nr AF  Hd Sec  Cyl  Hd Sec  Cyl    Start     Size ID
```

```
1 00    1    1     0  63  32  477      32  978912 83
2 00    0    1   478  63  32  509  978944   65536 82
3 00    0    0     0   0   0    0       0       0 00
4 00    0    0     0   0   0    0       0       0 00
```

这表明第一个分区从0柱面、1头、1扇区开始，一直延伸到包括477柱面、32扇区、63头在内的所有扇区。由于一个磁道有32个扇区，该硬盘有64个读/写头，所以这个分区是柱面大小的整数。fdisk缺省方式是按照柱面边界对齐分区的。所以该分区从最外面的0柱区向内扩展到478柱面，第二个交换分区从478柱面开始扩展到硬盘的最里面的柱面。

初始化过程中，Linux把硬盘的拓扑结构映射到系统中，它可以确定系统有多少个硬盘以及硬盘的类型。另外Linux还可以确定每个硬盘的分区数，这些信息是由gendisk_head表指针指向的gendisk数据结构表来表示的。在对像IDE这样的硬盘子系统初始化时，Linux为每个找到的硬盘产生一个代表该硬盘的gendisk数据结构。同时它会注册文件操作并在blk_dev数据结构中加入表项。每个gendisk数据结构有一个唯一的全设备号，并与该特殊块设备的主设备号一致。例如：SCSI硬盘子系统会建立一个gendisk表项(“sd”)，主版本号为8。这与所有SCSI硬盘设备的主版本号一致。图1-6-5给出了两个gendisk表项，第一个是SCSI硬盘子系统的，第二个是IDE硬盘子系统的，第二个表项的名称是“ide0”，指主IDE控制器。

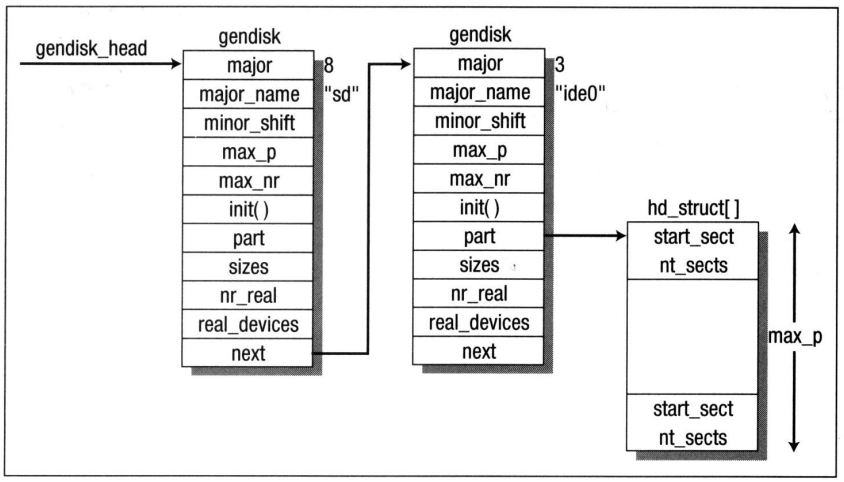


图1-6-5 硬盘表链

尽管硬盘子系统在初始化时建立gendisk表项，但它们只是在分区检查时才被Linux使用。每个硬盘子系统维护它自己的数据结构，从而允许其用主设备号和从设备号来映射到物理盘的某个分区上。无论块设备是通过缓冲区缓存机制还是文件操作进行读写，内核都会使用在特殊块设备文件中的主设备号把操作映射到相应的设备上。接着由每个设备驱动程序或内核子系统再把次设备号映射到实际的物理设备上。

1. IDE硬盘

现在Linux系统使用的大多数硬盘是IDE硬盘(Integrated Disk Eletronic)。IDE是一种磁盘接口，而不是像SCSI一样是一种I/O总线。每个IDE控制器可以最多支持2个硬盘：一个是主盘，另一个是从盘，硬盘的主从功能通常是由硬盘上的跳线来设定的。系统中的第一个IDE控制器被称为主IDE控制器，第二个被称为从IDE控制器。IDE可以管理大约3.3Mbps的数据传输量，

最大的IDE磁盘大小是538M字节，增强型IDE接口(EIDE)可以最大支持8.6G字节的硬盘，数据传输率增加到16.6Mbps，由于IDE和EIDE硬盘比SCSI硬盘便宜，所以大多数的现代PC都在主板上集成了一个或多个IDE控制器。

Linux按照硬盘所在的IDE控制器的次序来命名IDE硬盘，主控制器上主盘的名字是/dev/hda，从盘是/dev/hdb。/dev/hdc是从IDE控制器上的主盘。IDE子系统向Linux内核注册IDE控制器而不是硬盘。主IDE控制器的主标识是3而从IDE控制器的主标识是22。这表示如果一个系统有两个IDE控制器的话，IDE子系统会在blk_dev和blkdevs向量的索引值为3和为22的两个不同位置有两个不同的表项。IDE硬盘的特殊块设备文件能反映出这种编号方式，硬盘/dev/hda和/dev/hdb都连接在主IDE控制器上，因而它们的主标识为3。由于内核使用主标识作为索引值，所以任何对这些特殊块设备文件上的IDE子系统的文件或缓冲区缓存操作都会被定向到IDE子系统去。当有请求产生时，由IDE子系统计算出该请求是对哪个IDE磁盘的。为了计算出上述信息，IDE子系统使用特殊设备标识中的次设备号信息。在次设备号中包含能使IDE子系统把请求定向到正确的硬盘的正确分区的信息。/dev/hdb —— 主IDE控制器的从IDE硬盘的设备标识是(3, 64)，而该盘的第一个分区的设备标识是(3, 65)。

2. 初始化IDE子系统

IDE硬盘一直环绕着大部分IBM PC的历史。现在对这些设备的接口已经改变了，但这只是使得IDE子系统的初始化工作比以前更加复杂了。

Linux可以支持的最大IDE控制器的数目是4个，每个控制器由ide_hwifs向量中的一个ide_hwif_t数据结构来表示。每个ide_hwif_t数据结构包含两个ide_drive_t数据结构，分别用于支持主从IDE驱动器。在IDE子系统初始化时，Linux在系统CMOS存储器中查找当前硬盘的信息。CMOS存储器是由电池供电的，即使在PC关机时也不会丢失信息。CMOS存储器的位置是由系统的BIOS指定的，它可以通知Linux当前系统中找到的IDE控制器和驱动器。Linux从BIOS中读取硬盘的结构信息，并使用这些信息来建立该驱动器的ide_hwif_t数据结构。大多数的现代PC使用像Intel的82430 VX那样的芯片组，它们都包含PCI的EIDE控制器，IDE子系统使用PCI BIOS的回调功能来定位系统中的PCI(E)IDE控制器，然后它为这些芯片组调用PCI的特殊询问例程。

一旦发现了一个IDE接口或控制器，它的ide_hwif_t数据结构就被建立起来，以反映控制器及其相连的硬盘状态。在操作过程中，IDE驱动程序会向I/O存贮空间的IDE命令寄存器写入命令。主IDE控制器的控制和状态寄存器的缺省I/O地址是0xF0到0xF7。这些地址是遵照早期的IBM PC的习惯而设定的，IDE驱动程序会向Linux的缓冲区缓存和VFS文件系统注册所有的控制器，并把它们分别加到blk_dev和blkdevs中去。IDE驱动程序也会请求相应的中断控制，由于遵循IBM PC的习惯，主IDE控制器的中断号为14，从IDE控制器的中断号是15。但它们也像IDE的所有参数一样可以由内核的命令行选项来配置的，IDE驱动程序会在gendisk列表中为启动时找到的每个IDE控制器增加一个gendisk表项。这个表接着会被用于查找启动时发现的所有硬盘的分区表信息。分区表检查程序知道每个IDE控制器可以连接两个IDE硬盘。

3. SCSI硬盘

SCSI(小型计算机系统接口)总线是一种高效的对等数据总线，每条总线最多支撑8个设备(包括一个或多个主动设备)。总线上的每个设备都有一个通常由硬盘上的跳线设定的唯一标识。数据可以以同步或异步的方式在总线上的任何两个设备之间进行传送，在使用32位总线时数

据的最大传输率可以达到40M字节/秒。SCSI总线在设备间既可以传递数据又可以传递信息，总线上创始者与目标设备间的一个交易可以最多包括 8个不同的阶段。可以从总线的 5个信号中获得当前SCSI总线的阶段。这八个阶段是：

- 总线空闲 无设备控制总线，也没有交易正在进行。
- 仲裁 SCSI设备通过在地址引脚上设置它的 SCSI标识来申请获得SCSI总线的控制权。SCSI标识号最高的设备获得控制权。
- 选择 当一个设备成功地通过仲裁获得了SCSI总线的控制权后，会向这个SCSI请求的目标设备发信息，通知目标设备它要发送命令。创始者是通过在地址引脚上设置目标设备的SCSI标识来完成通知的。
- 再选择 SCSI设备可以在处理请求时断开连接，接着目标设备会重新选择创始者。但并不是所有的SCSI设备都支持这一阶段。
- 命令 创始者可能会向目标设备发送6字节、10字节或12字节的命令。
- 数据输入/输出：在这一阶段中，创始者与目标设备间正在交换数据。
- 状态 所有命令完成之后才进入这一阶段，在该阶段目标设备可以向创始者发送状态字节以显示命令成功与否。
- 消息输入/输出 这一阶段用于在创始者、目标设备间传送额外的消息。

Linux的SCSI子系统由两个基本部分组成，每一部分由一个数据结构来表示。

- 主动设备 一个SCSI主动设备是一部分物理硬件 —— SCSI控制器。NCR810 PCI SCSI控制器就是一种SCSI主动设备。如果一个Linux系统可以有同类型的一个以上的 SCSI控制器，那么每个实例都会由一个独立的 SCSI主动设备来表示。这意味着 SCSI设备驱动程序可以控制一个以上的控制器实例， SCSI设备几乎总是SCSI命令的创始者。
- 设备 最普通的SCSI设备是SCSI硬盘，但SCSI标准还支持几种设备类型：磁带、 CD-ROM以及通用SCSI设备。SCSI设备几乎总是SCSI命令的目标设备，这些设备必须按不同的方式来处理。如对 CD-ROM和磁带这种有可移动介质的设备， Linux必须检测介质是否被取出了。由于不同的磁盘类型有不同的主设备号，所以 Linux可以把不同的块设备请求定向到相应的SCSI类型的设备上去。

(1) 初始化SCSI子系统

初始化SCSI子系统十分复杂，反映出了 SCSI总线和SCSI设备的动态特征。 Linux在启动时初始化SCSI子系统，它先查找系统中的所有 SCSI控制器，然后再检测每条 SCSI控制器的SCSI总线，找出连接的所有设备。最后它初始化这些设备，使得它们通过普通文件操作和缓冲区缓存设备操作对Linux内核的其它部分是可用的。初始化过程分四个阶段：

第一阶段：Linux找出在编译内核时安装的那些用于控制硬件的 SCSI主动设备适配器或控制器，每个在内核中安装的 SCSI主动设备都在 builtsin_scsi_hosts向量中占据一个Scsi_Host_Template表项。Scsi_Host_Template数据结构包含指向执行特殊的 SCSI主动设备动作的例程的指针，这些动作包括检测有哪些 SCSI设备附着在这个SCSI主动设备上。这些例程在SCSI子系统，自我配置时被调用并且它们也是支持这种类型主动设备的 SCSI设备驱动程序的一部分。每一个被检测的 SCSI主动设备，为每个依附于它的真实的 SCSI设备，都在活动SCSI主动设备的SCSI_Host列表中增加一个scsi_Host_Template数据结构。每个被检测出来的主动类型设备的实例都由 scsi_hostlist列表中的一个Scsi_Host数据结构来表示。例如一个系统

如果有2个NCR810 PCI SCSI控制器，则它在列表中会有两个Scsi_Host表项，每个控制器一个。每个由Scsi_Host_Template指向的Scsi_Host结构都代表着它对应的设备驱动程序。

在找到了所有的SCSI主动设备之后，SCSI子系统要确定每个主设备的总线上连接着哪些SCSI设备。SCSI设备在0~7之间进行编号，每个SCSI设备在它所连接的总线上都有唯一的设备号或SCSI标识。SCSI标识通常是由设备上的跳连来设置的。SCSI初始化程序通过向SCSI总线发送TEST_UNIT_READY命令来查找该总线上的设备，一个设备作出响应时，SCSI初始程序通过向它发送ENQUIRY命令获得它的标识。从标识中Linux可以获得厂商名、设备模式和修正名。SCSI命令由Scsi_Cmnd数据结构表示，通过调用该SCSI主动设备的Scsi_Host_Template数据结构中的设备驱动程序，例程可以把标识传送给该主动设备的设备驱动程序。每个找到的SCSI设备由一个Scsi_Device数据结构表示，每个结构中都包含有指向父结点 Scsi _Host结构的指针，所有的Scsi_Device数据结构都被加入到scsi_devices表中。图1-6-6给出了主要的结构间的关系。

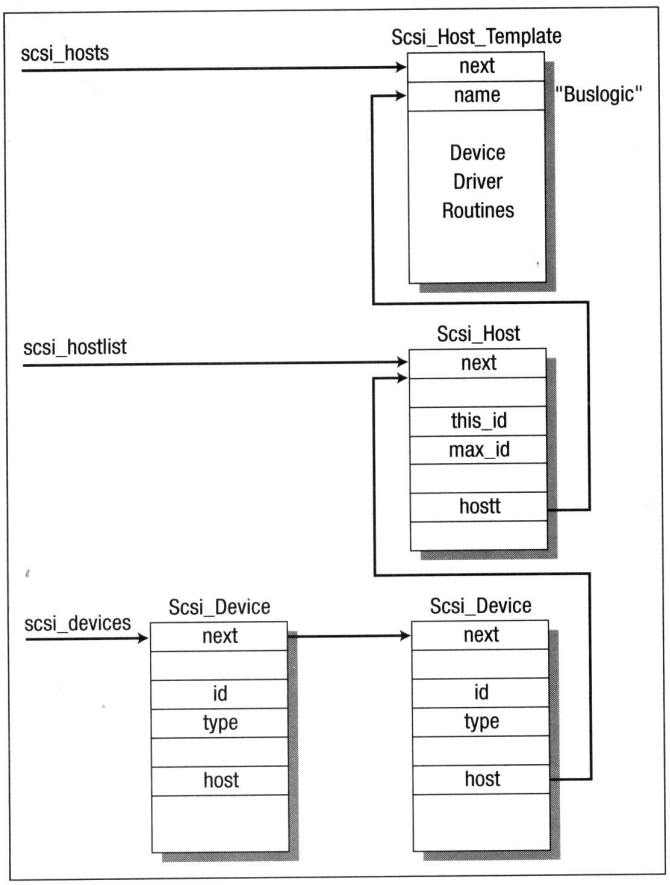


图1-6-6 SCSI数据结构

总共有4种SCSI设备类型：磁盘、磁带、CD-ROM和通用SCSI设备，每种SCSI类型都在内核中以不同的主块设备类型进行注册。然而如果系统中有一种或多种 SCSI设备类型，那么每种设备只会注册它们自己。每种 SCSI类型都维护它们自己的设备表，它使用这些表格把内核的块操作定向到正确的 SCSI设备驱动程序或 SCSI主动设备上。每种 SCSI设备类型由一个

Scsi_Device_Template数据结构来表示。Scsi_Device_Template数据结构包含对应的SCSI设备类型的信息以及用于执行各种操作的例程的地址。 SCSI子系统使用这些模板为每种类型的SCSI设备调用相应的例程，换句话说就是，如果SCSI子系统要连接一个SCSI硬盘设备，它会调用SCSI硬盘类型的连接例程。如果有一个或多个某种类型的 SCSI设备被检测出来了，那么该种类型对应的Scsi_Type_Template数据就会被加入到scsi_devicelist表中。

SCSI子系统初始化的最后阶段是为每个注册的 Scsi_Device_Template调用结束函数。对SCSI硬盘类型来说，它会使所有找到的 SCSI硬盘旋转起来，记录它们的磁盘参数；它还会把代表所有SCSI硬盘的gendisk数据结构加到图1-6-5中给出的硬盘链表中。

(2) 传送块设备请求

一旦Linux初始化了SCSI子系统，SCSI设备就可以使用了。每个活动的 SCSI设备类型都向Linux内核注册自己，以便Linux能够把块设备请求定向给它。这些请求包括通过 blk_dev结构的缓冲区缓存请求和通过 blkdevs的文件操作请求。用有一个以上 EXT2文件系统分区的SCSI硬盘的驱动程序作为例子，让我们看一下当安装了多个 EXT2分区时，内核缓冲区请求是如何被定向到正确的SCSI硬盘上的。

每个从SCSI硬盘分区中读一块数据或写入一块数据的请求都会使 blk_dev向量中SCSI硬盘的current_request列表增加一个新的请求数据结构。如果请求表正在被处理的话，缓冲区缓存机制不需要做其它的工作；否则，它要通知 SCSI硬盘子系统处理请求队列。系统中的每个SCSI硬盘由一个Scsi_Disk数据结构来表示，这些数据结构被保存在 rscsi_disks向量中，使用SCSI硬盘分区次设备号的一部分来进行索引。例如： /dev/sdb1的主设备号是8，次设备号为17；因此它对rscsi_disks向量的索引值为1。每个Scsi_Disk数据结构都有一个指向代表该设备的Scsi_Device数据结构的指针，而 Scsi_Device反过来又包含有指向它所在的主动设备的Scsi_Host数据结构的指针。从缓冲区缓存来的 request数据结构先被转化成要发送给 SCSI设备的SCSI命令的Scsi_Cmd数据结构，然后这些Scsi_Cmd结构被加入到代表该设备的Scsi_Host结构的队列中。每个SCSI驱动程序一旦读/写了适当数据块后就会处理这些命令。

6.2.6 网络设备

网络设备是与Linux网络子系统相关的，用于发送/接收数据报文的实体。它通常是像以太网卡那样的物理设备，但也可能是软件；如用于向自己发送报文的回送 (loopback)设备。每个网络设备由一个device数据结构来表示，在内核启动、网络初始化时，网络设备驱动程序向Linux注册它们所控制的设备， device数据结构中包括该设备的信息以及允许各种支持的网络协议使用设备服务的函数集合的地址。这些函数多半是与使用网络设备传输数据相关的。设备使用标准的网络支持机制，把接收到的数据上传到适宜的协议层中去。所有的网络数据 (报文)的发送和接收都是由sk_buff数据结构来表示，这些数据结构非常灵活，允许网络协议头可以很容易地被加入或删除。网络协议层如何使用网络服务以及它们是如何使用 sk_buff数据结构正向和反向传送数据的？这些都在第8章有详细的介绍。本部分主要考虑的是 device数据结构以及网络设备是如何被发现和初始化的。

device数据结构包含下列关于网络设备的信息：

- 名称 不像块设备和字符设备那样可以通过 mknod命令建立它们的特殊设备文件，网络设备的特殊设备文件是在系统的网络设备检测和初始化时建立起来的。网络设备的名称

是标准的，每类名称代表一种设备类型。同一类型的多个设备以 0开始向上编号，所以以太网设备可以被称为 /dev/eth0、/dev/eth1、/dev/eth2等等。一些通用的网络设备名如下所示：

/dev/ethN	以太网设备
/dev/slN	SLIP设备
/dev/pppN	PPP设备
/dev/lo	Loopback设备

- 总线信息　是设备驱动程序用于控制设备的信息。 irq号是该设备正在使用的中断号。而基地址(base address)是设备的控制和状态寄存器在 I/O空间的起始地址， DMA通道(channel)是网络设备正在使用的 DMA通道号。所有的这些信息都是在系统启动时设备初始化过程中设定的。
- 接口标志　它们描述了网络设备的特性和能力：

IFF_UP	接口正在运行
IFF_BROADCAST	在设备中的广播地址是有效的
IFF_DEBUG	打开设备调试
IFF_LOOPBACK	这是一个回送设备
IFF_POINTTOPOINT	这是点到点链接(SLIP和PPP)
IFF_NOTRAILERS	不允许网络追踪
IFF_RUNNING	分配资源
IFF_NONRP	不支持APP协议
IFF_PROMISC	设备处于混杂接收方式，它能接受所有的报文，而不考虑报文的目的地址
IFF_ALLMULTI	接收所有的IP多路广播帧
IFF_MULTICAST	可以接收IP的多路广播帧

- 协议信息　设备用于描述如何支持网络协议层的信息。
- mtu　不包括要加的所有链路层头时，网络可以传输的最大报文大小。该值由 IP等协议层使用，用于选择发送报文的大小。
- 家族　家族用于表示设备能够支持的协议家族。所有 Linux网络设备的家族是 AF_INET —— Intenet地址家族。
- 类型　硬件接口类型描述该网络设备连接的介质类型。 Linux网络设备支持很多种不同的介质类型，包括：以太网、 X.25、令牌环、 SLIP、PPP和Apple Localtalk。
- 地址　device数据结构包含很多与该网络设备有关的地址，如该设备的 IP地址等等。
- 报文队列　它是等待由该网络设备发送的由 sk_buff结构表示的报文的队列。
- 支持函数组　每个设备都有一组标准的例程，作为设备链路层接口的一部分可以由协议层来调用。它们包括建立数据帧、传送数据帧的例程以及增加标准帧头和收集统计信息等例程。统计信息可以由 ifconfig命令看到。

初始化网络设备

网络设备驱动程序可以像其它的 Linux设备驱动程序那样，安装在 Linux内核中。每种可能的网络设备都由 dev_bast表指针指向的网络设备表中的 device数据结构来表示。如果网络层

需要设备执行某些特殊工作的话，它可以调用记录在 device数据结构中的某一个网络设备服务例程。但是每个device数据结构最开始只包括初始化或探测例程的地址。

网络设备驱动程序需要解决两个问题：第一个问题是并不是安装在内核中的所有网络设备驱动程序都能找到要控制的设备。第二个问题是无论以太网的下层设备驱动程序是什么，它们在系统中的名称总是/dev/eth0、/dev/eth1……。网络设备“缺少”的问题很容易解决，在调用每个网络设备的初始化例程时，它会返回一个状态码显示它是否找到其驱动的控制器实例。如果驱动程序没找到任何设备，那么它在由 dev_base指向的device列表中的表项就被删除了。如果驱动程序找到了设备，它会用该设备的信息填充 device数据结构的其余部分，并在其中写入网络设备驱动程序的支持函数的地址。

对第二个动态地将以太网设备分配给标准的特殊设备文件 /dev/ethN的问题，Linux用一种更高明的办法解决了。在设备表中共有 8个标准表项，第一个对应 eth0，第二个对应eth1。并以此类推。8个表项中的初始化例程是完全相同的。 Linux轮询安装在内核中的每个以太网设备驱动程序，直到有一个驱动程序找到了它控制的设备。一旦一个驱动程序找到了它的以太网设备，就会填充它现在占用的 ethN device数据结构。并且在网络设备初始化时，驱动程序会初始化它控制的物理硬件，找出设备使用的中断号、 DMA通道号等信息。如果设备驱动程序找到了它控制的网络设备的多个实例的话，就会占用多个 /dev/ethN device数据结构。一旦8个标准/dev/ethN特殊设备文件都被分配了， Linux就不会再检测其它的以太网设备了。